

The Inspired Handbook

Reading the image, cutting the object out, shopping the cut.

Inspired

This PDF edition is generated from the same Markdown chapters as the HTML edition. It is set in Helvetica and Courier with WinAnsi encoding, so the diagrams are transliterated to ASCII and lines are set ragged-right without hyphenation. For the diagrams as drawn, read the HTML edition.

This is the book about why Inspired is built the way it is.

The quickstart tells you which endpoint to call. The API reference tells you what the fields mean. Neither tells you why a read takes minutes and cannot be made to take seconds, why extraction is deliberately sequential when parallel would be four times faster, why your request is charged before any work happens, or why a 404 refuses to say whether the read exists.

Every one of those is a decision with a reason, and knowing the reason is the difference between integrating against this platform and fighting it.

Nine chapters, about an hour:

1. What this actually does -- the three moves, and what is not in them.
2. The pipeline -- stage by stage, and why the slow part is slow.
3. Durability -- the Workflow, the picker that never blocks, and what happens when a phone locks.
4. Focus domains -- one pipeline, a parameterised subject.
5. Keys and allowance -- charging at the start, and why re-minting gives you nothing.
6. The two surfaces -- HTTP and MCP, and why neither can outrun the other.
7. Reading a result -- the fields, the ordering, and the images.
8. Failure -- every error, whether it was charged, and whether to retry.
9. The honesty contract -- the words the product may use, and why the vocabulary is a technical constraint rather than a style guide.

Written against the platform deployed at inspired.jetskibay.com in September

1. Where a behaviour might change, the chapter says so.

Contents

- 1. What this actually does
- 2. The pipeline
- 3. Durability
- 4. Focus domains
- 5. Keys and allowance
- 6. The two surfaces
- 7. Reading a result
- 8. Failure
- 9. The honesty contract

1. What this actually does

Three moves, in order, on one photograph.

Read it. One vision call names every separately purchasable object in the frame and describes what is genuinely visible of each: colour, material, shape, the details it can make out, and -- critically -- which objects hide which. Up to six objects. A photograph with nothing clearly shoppable returns zero.

Cut each one out. One image-editing call per object produces an isolated e-commerce product shot on plain white: the occluding objects deleted, the parts they hid rebuilt in the same material, the finish and silhouette held to what the photo actually showed.

Shop the cut. Each cutout -- not the original photograph -- goes to visual search in product mode. Up to ten price-led results per object, with retailer, title, price, image and link.

Then a mood board: the cutouts composed into one square image.

Why the middle step exists

The naive version of this product skips it: send the photograph to visual search and return what comes back. That fails for a reason worth understanding, because it is the reason the whole architecture is shaped the way it is.

A photograph of a person in an outfit is one image containing six products. Ask visual search about it and you get results for the photograph -- the same photograph on another site, a similar pose, a stock image of a model. The one thing you cannot get is the skirt.

So the cut has to happen first, and it has to happen per object. That single requirement produces: a reader that must describe occlusion, an editor that must rebuild what it deletes, six sequential extraction calls instead of one, and a read measured in minutes. Everything in this book follows from it.

What is not in here

No verification. Nothing re-checks a cutout against the original photograph. The pipeline that produced the cutout is the only thing that has an opinion on whether it is right, and it is not asked twice.

No similarity score. Nothing scores how close a retailer result is to the cutout. The results are in the order visual search returned them, price-led, and that order is the only ranking signal in the response.

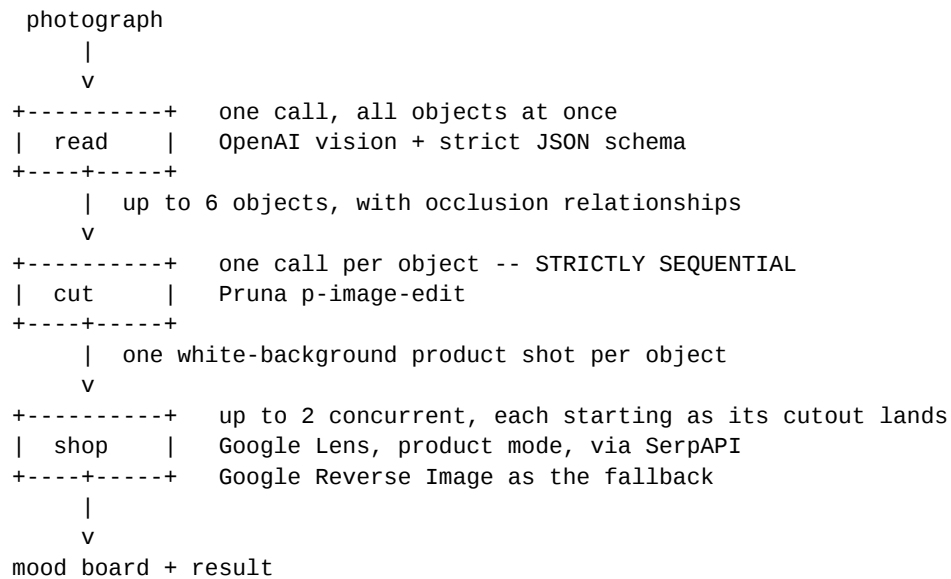
No brand identification. A brand is named only when a brand mark is clearly legible in the photograph. Otherwise the description stays descriptive, because a guessed brand is worse than no brand.

No image fetching. The platform accepts image bytes or a public Pinterest pin URL. There is no parameter that takes an arbitrary image URL, deliberately: such a parameter would make this an open image proxy pointed at anyone's server.

No speed. Chapter 2.

Those five absences are load-bearing. Every one of them could be built, and every one of them would change what the product is allowed to claim. Chapter 9 is about what it is allowed to claim as it stands.

2. The pipeline



The read

One call, every object at once, and this is not a cost optimisation -- it is what makes the occlusion relationships possible. A model asked about the whole frame can say the vest covers the shirt; a model asked about each object separately cannot, because it never sees them together.

The output is a strict schema: `additionalProperties: false`, every property required. A model cannot skip the awkward fields. An empty list of objects is expressible and is the right answer for an unshoppable photo; a partial object is not expressible at all.

The cut, and why it is sequential

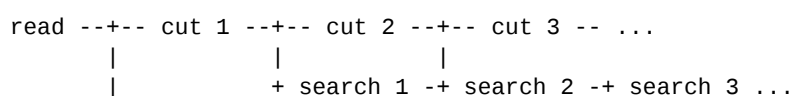
One extraction call per object, one at a time, never in parallel.

Parallel extraction is four times faster and measurably worse. Every call receives the same source photograph, and a model extracting the shirt while another extracts the vest has no way to know which garment the request meant -- the failure is not a crash, it is a shirt that comes back wearing the vest, or a vest with a shirt collar showing through it. Sequencing does not fix that on its own, but it is what lets each call carry the full occlusion story for one object without competing for it.

This is the slow part. Six objects is six calls, each seconds to tens of seconds. It is why a read is minutes.

The shop, and why it overlaps

Search does not wait for extraction to finish. Each object's search starts the moment its own cutout lands, with up to two in flight. So the timeline is:



The tail of searches finishes shortly after the last extraction. Total time is roughly `read + (objects x cut) + one search`, not `read + cuts + searches`.

Sparse results fall back from Google Lens product mode to Google Reverse Image. Lens's ordering is preserved as returned -- there is no re-ranking step, because there is nothing to re-rank with.

Retailer curation

Two gates, and they exist because of one complaint.

Visual search returns whatever the index holds, and ultra-fast-fashion and dropship marketplaces relist a single garment under dozens of titles. Left alone, one host takes every slot in a row.

So: those hosts are dropped outright, and no retailer may lead with more than two results for one object. The cap matters more than the block list. The complaint was never that a particular marketplace appeared -- it was that it was all there was. A spread of retailers is what makes a row read as shopping.

Where each stage runs

Cloudflare Workers throughout, with a Cloudflare Workflow as the durable job (chapter 3). The reading call goes to OpenAI; extraction to Pruna p-image-edit; search to SerpAPI's Google Lens. Uploads, cutouts and results live in Cloudflare R2.

Every stage writes a private evaluation bundle -- source image, prompt, schema, provider responses, generated assets, final output, credentials excluded. Nothing reads those bundles today. They are a corpus waiting for a harness, and calling them a quality loop would be a lie.

3. Durability

A read is minutes long, and the thing that started it is usually a phone that is about to lock.

So the read is not a request. It is a durable job with a stable id, and everything that talks to it -- the app, the API, an agent over MCP -- is a client reconnecting to it.

One Workflow step per unit of paid work

The job is a Cloudflare Workflow, and the step boundaries are drawn around money, not around code structure:

1. analyze look -- resolve the source, one vision call, one upload to the extraction service.
2. wait for piece selection -- a 90-second grace window.
3. extract piece-N -- one extraction call. One step per object.
4. search piece-N -- that object's searches, overlapping the next extraction.
5. finish search -- mood board, stored result, private archive.

Drawn this way, a failure retries one paid call. Drawn any coarser -- one step for "extract everything" -- a single failed extraction replays the vision call and every finished extraction with it, and the retry costs more than the original attempt.

The verdict that must not be retried

Some failures are stable: a photograph with no clothing in it, a look where every extraction failed.

Retrying those buys two more identical answers.

They raise a non-retryable error, so the job fails once and reaches the person. The distinction -- this will differ next time versus this is the answer -- has to be made by the stage that knows, because the retry machinery cannot guess.

The picker is a choice, never a gate

The consumer app offers a choice of which objects to shop, and this is where durability gets interesting.

The 90-second window is a grace period, not a barrier. If nobody answers, the pipeline continues with every object it spotted. A locked phone still returns to a finished edit.

A choice that arrives later still counts: the selection is stored and re-read before each extraction, so someone coming back at minute three can still drop objects that have not started. An object already being worked on completes.

The API does not show a picker at all. An API caller asked about the whole image, so the job runs with selection disabled.

What a client is allowed to be

Because the job holds the state, a client can be almost nothing:

- The browser stores the pending upload locally and starts the job under a stable id, then reconnects by polling it. Locking the screen, switching apps or reloading the tab changes nothing.
- The iPhone share extension starts the job inside the share sheet and stores it in the App Group, so opening the app lands directly in progress or results.
- An API caller polls GET /api/v1/jobs/{id}.
- An agent calls get_visual_search.

Four clients, one job, no session.

Reusing an id is safe

Starting a read with an id that already exists returns the existing job rather than starting a second one -- unless the previous attempt saved its upload and then died before creating the Workflow, in which case the same id repairs the gap.

That is what makes a client's retry safe. A dropped response on a start call does not cost a second request, because the id is the idempotency key.

Retention

The uploaded photograph, the cutouts and the result live in the private job bucket for one day, which is the window a suspended browser needs to reconnect. A signed-in person's finished result is copied into their private library before that cleanup, which is what makes History and Gallery possible.

An anonymous read belongs to nobody and is gone with the job. Evaluation bundles are not covered by the one-day cleanup.

4. Focus domains

Inspired began as one pipeline with one subject: the clothes on a person. The platform sells the same three moves against any class of object, so the subject had to become a parameter.

What a domain actually owns

A domain owns exactly two things:

1. The reading prompt, with its JSON schema.
2. The extraction instruction.

Everything downstream -- the extraction service, the searches, the mood board, the durable job -- was already subject-agnostic and was not touched. That is why adding a domain is a five-line change: the subject nouns, the name examples, and what to exclude.

domain	Subject	Aliases
-----	-----	-----
fashion (default)	Garments, shoes, bags, jewellery on a person	clothing, clothes, outfit, ap
furniture	Sofas, chairs, tables, lighting, rugs, decor	interior, home
devices	Phones, laptops, audio, cameras, appliances	tech, electronics
beauty	Skincare, makeup, fragrance, haircare, tools	--
everything	Any distinctly purchasable object	any, all

Fashion is frozen on purpose

fashion returns the exact prompt, schema and extraction instruction the consumer app has always used -- character for character, asserted by the test suite. Every other domain is generated from a shared generic profile.

This is not sentimentality. The fashion prompt was tuned against real photographs over many iterations, and the consumer app's output quality is the business. Generalising the pipeline could not be allowed to regress it, so the old path is pinned by a test and the new path is built beside it.

The fashion extraction instruction also lives in a different file from the others -- beside the extraction call it was tuned against -- and the domain declares that by saying it has no prompt of its own. Where code lives is sometimes an argument about what it belongs to.

What changed when the subject generalised

The vocabulary, and almost nothing else.

layer (outer, mid, base, bottom, footwear, accessory) becomes placement (foreground, midground, background, on-top-of, inside, standalone), because a photographed room occludes objects the same way a vest occludes a shirt -- it just cannot call a lamp a base layer.

hardware becomes details. fabric_finish becomes material_finish. silhouette becomes form. And front_opening -- whether a top layer hangs open and what shows through the gap -- has no general equivalent, so it was dropped rather than faked. A coffee table has no front opening, and a field that

always says "not applicable" trains a model to ignore the schema.

Every structural rule transferred untouched: lead with the deletion, rebuild what was hidden, hold the finish and the shape, treat uncertain details as a ceiling rather than an instruction, enumerate the objects that must not appear, and return an empty list rather than a guess.

An unknown domain is an error

domain=submarines is a 400, refused before anything is stored or charged. It does not fall back to clothing.

A silent fallback would take a typo, spend a request, and return a confident description of an outfit nobody asked about. The caller would have no way to tell that from a correct answer. Refusing is cheaper for everyone.

focus

Up to 160 characters of your own words, appended to the reading instruction as one clause:

```
domain=everything focus=kitchen appliances only
```

Control characters are stripped and the text is collapsed to a single line, so it stays a clause inside the request rather than becoming a second instruction block. It is built specifically not to be a prompt override, and treating it as one will not work.

Use it to narrow within a domain -- which is what makes everything plus focus the escape hatch for a category nobody has named yet.

5. Keys and allowance

A trial key is ten requests and twenty-four hours. Every design decision around it follows from one fact: a single call behind that key costs a vision call, an extraction per object, and two searches per object. Ten of them is a real bill.

The key exists in one email

POST /api/v1/keys with an address mints a key, mails it, and returns the prefix and the allowance -- never the key. Only its SHA-256 is stored.

So there is no "show me my key again", and losing one is not a disaster: re-minting for the same address revokes the old key and carries the remaining allowance over. Losing a key costs nothing. Asking twice buys nothing.

That property is why there is no rate-limit exemption needed for support cases and no way to farm allowance by re-minting. It is capped anyway -- five mints per address per day, ten per network, three per minute -- but the carry-over is what makes the cap uninteresting to attack.

The address itself is stored only as a keyed digest, along with the network the mint came from. They exist to enforce those caps, not to build a mailing list, and the keying means a database leak does not yield addresses to a wordlist.

Charged at the start

A request is charged when a read starts, not when it finishes.

The obvious alternative -- charge on success -- has a hole: ten concurrent calls against one remaining request all read a count below the limit and all proceed. The guard therefore lives in the SQL update itself:

```
UPDATE api_key
  SET request_count = request_count + 1, last_used_at = ?
 WHERE id = ?
    AND revoked_at IS NULL
    AND expires_at > ?
    AND request_count < request_limit
```

If that affects zero rows, the key had nothing left. Two simultaneous calls cannot both win, because the database decides, not the application.

Refunded only when we broke it

The charge comes back for a platform failure -- a 5xx, an upstream outage. It does not come back for:

- An empty result. A photograph with nothing shoppable in it is a successful read. The vision call happened and cost money. This is the one that surprises people, and it is correct.
- Your malformed request, if it got far enough to be charged. An unknown domain is rejected before the charge, so that one is free.
- A read you abandon. The work runs whether or not you collect it.

A refund can never take the count below zero, so a duplicated refund cannot mint free requests.

What is free

Polling. Key status. Revocation. The domain catalogue. list_focus_domains over MCP.

Polling in particular is free on purpose. A metered poll would push callers into longer intervals and worse latency for no benefit -- answering a poll costs almost nothing, and the alternative is a product that feels slower than it is.

The unmetered key

An environment-configured key authenticates as an unmetered root tier. It predates the key table and keeps existing integrations and the repository's own examples working. It is a deployment secret, not something a caller can obtain.

One read, one request

A trial key is ten photographs, not ten API calls. While you are building, run one read, save the JSON, and develop against the saved payload -- the shape does not change. The single fastest way to empty a key is a retry loop around a read that is still running.

6. The two surfaces

The same three capabilities, twice.

	HTTP	MCP
-----	-----	-----
Where	/api/v1	/mcp
For	anything that is not an agent	an agent runtime
List domains	GET /api/v1/domains	list_focus_domains
Start a read	POST /api/v1/jobs	start_visual_search
Collect a read	GET /api/v1/jobs/{id}	get_visual_search
Synchronous read	POST /api/v1/inspire	--
Key management	/api/v1/keys	--
Auth	x-api-key or bearer	bearer or x-api-key

Neither can outrun the other

The MCP tools do not reimplement anything. `start_visual_search` builds the multipart request that `POST /api/v1/jobs` already validates and calls it; `get_visual_search` does the same for the status route.

That is deliberate, and it is worth stating as a property: an MCP caller cannot obtain a read an HTTP caller could not. One code path validates the domain, one charges the quota, one stamps the job's owner. A second implementation of any of those would eventually disagree with the first, and the disagreement would be a security bug rather than an inconsistency.

What MCP changes

Two things, both about the shape of a result rather than its content.

Images become links. The HTTP job response embeds every cutout and the mood board as base64 data: URLs -- correct for a client writing files to disk, ruinous for a model's context window, where a six-object result is megabytes of tokens. Over MCP each cutout is an HTTPS URL and the mood board is described rather than inlined, with a pointer at the HTTP route for the bytes.

Errors become results. A failed tool call comes back as a successful JSON-RPC response with `isError: true` and a readable reason, not as a JSON-RPC error. A model can read a tool error and decide what to do next; a JSON-RPC error is the runtime's problem and usually never reaches the model at all.

The transport, honestly

Streamable HTTP, JSON responses only, one pinned protocol revision. What is not implemented, and answers 405 or -32601 rather than pretending:

- GET /mcp -- there is no SSE stream to open.
- DELETE /mcp -- there is no session to terminate.
- JSON-RPC batching -- removed from the protocol.
- resources/list, prompts/list -- only tools is declared in initialize.

A half-built stream that accepts a connection and never sends anything is worse than an honest

refusal, because it fails at runtime in a client rather than at configuration time.

Discovery is authenticated. There is no anonymous tools/list: the credential is checked on every request including initialize. A key with a spent allowance can still list tools and collect a read it already paid for -- exactly as polling stays free over HTTP -- and only start_visual_search is refused.

Why the synchronous route still exists

POST /api/v1/inspire runs the whole read and returns the whole result in one response. It is genuinely convenient in a terminal, and that is the entirety of its purpose. It holds the connection for minutes; most HTTP clients give up first and every serverless caller does.

It is documented, it is supported, and it is not the integration path. If you find yourself adding a retry policy around it, you wanted /api/v1/jobs.

7. Reading a result

```
{
  "requestId": "3f0c8a5e-...",
  "status": "complete",
  "domain": "furniture",
  "focus": "",
  "summary": "Found 3 clearly visible furniture items.",
  "items": [
    {
      "id": "piece-1",
      "simple_name": "floor lamp",
      "full_description": "Slim brass floor lamp with a white cone shade",
      "gridCell": 1,
      "imageDataUrl": "data:image/webp;base64,...",
      "candidates": [
        { "title": "Brass arc floor lamp", "link": "https://...", "price": "EUR149", "imag
      ]
    }
  ],
  "grid": { "imageDataUrl": "...", "mimeType": "image/webp", "cells": [ ... ] },
  "metadata": { "domain": "furniture", "visionModel": "...", "searchMode": "reverse_image"
}
```

The fields that carry meaning

simple_name is the shopping word: skirt, floor lamp, over-ear headphones. Two objects in one photograph can share it.

full_description is what the reader could actually see. On a half-hidden object it describes the visible parts and says so rather than guessing the rest. Read it as evidence, not as a product title.

gridCell is the object's position in the mood board, one-based.

candidates is ordered, and the order is not yours to improve. It is visual search's own product-mode ordering, price-led. There is no similarity score in the response because there is no similarity score anywhere in the pipeline -- re-sorting by your own guess at closeness makes the result worse and gives your users a ranking you cannot explain.

items is present and empty when the photograph had nothing clearly enough visible to shop for. That is a successful read (chapter 5).

What is missing, and is meant to be

No confidence value. No bounding boxes. No similarity score. No brand field.

Each of those is a number people would put in a UI, and none of them exists in the pipeline. Publishing a placeholder -- a confidence of 0.9 because it looked about right -- would be worse than the absence, because the absence at least tells you the truth about what was computed.

The images, and which ones expire

Three kinds, and they behave differently:

Cutouts. `items[].imageDataUrl`. Over HTTP, a base64 data: URL. In a job result they may instead be a path, `/api/inspire/jobs/{id}/assets/piece-N.webp` -- an unguessable per-job path, cached privately, valid for the job's one-day retention. Over MCP you always get the absolute URL form.

The mood board. `grid.imageDataUrl`, a composed square WEBP. Over MCP it is described rather than inlined.

Candidate images. `candidates[].imageUrl`. Hosted by the retailers and by Google. Not ours, not stable, and they will 404 eventually. Re-host anything you need to keep.

Progress, while it runs

```
{
  "status": "running",
  "progress": {
    "stage": "extract",
    "title": "Cutting out the floor lamp",
    "items": [
      { "id": "piece-1", "simple_name": "floor lamp", "status": "extracting" },
      { "id": "piece-2", "simple_name": "sofa", "status": "queued" }
    ]
  },
  "pollAfterMs": 3000
}
```

The object list is known after the reading call, which is the first thing that finishes. Render it immediately. A named placeholder per object, filling in one at a time, reads as progress; a spinner held for three minutes reads as a hang. This is the single biggest difference between the API feeling slow and feeling broken, and it costs nothing to get right because the data is already there.

`pollAfterMs` is a server-side hint. Honour it rather than hard-coding an interval: polling is free, so there is nothing to gain by going faster and nothing to lose by asking.

8. Failure

Status	code	Charged?	Retry?
-----	-----	-----	-----
400 bad multipart	--	no	after fixing the body
400 unknown domain	--	no	after fixing the domain
401	missing_api_key	no	no -- send the credential
401	invalid_api_key	no	no
401	revoked_api_key	no	no -- mint another
401	expired_api_key	no	no -- mint another
404 on a read	--	no	no -- check the id
405	--	no	no -- wrong method
413	--	no	no -- smaller body
429	quota_exhausted	no	no
429 mint limit	--	no	later
503	api_not_configured	no	later, platform-side
503	--	no	later, platform-side
200 status: "failed"	--	refunded	once
200 items: []	--	yes	no -- that is the answer

The three that get handled wrongly

quota_exhausted is terminal. The key has nothing left. Retrying will not help and minting another key will not help -- re-minting carries the remaining balance over rather than resetting it (chapter 5). Stop, and tell whoever is watching.

items: [] is a success. The reader is built to return nothing rather than invent an object it cannot see. Handle it as an answer -- "nothing clearly shoppable in this photo" -- and note that it did cost a request. Retrying the same image produces the same empty answer for another request.

A 404 on a read will not tell you why. It means either no such read exists or the read belongs to another key, and the response is deliberately identical for both. A request id is a UUID; confirming one exists would leak it. There is nothing to distinguish, so do not build a branch that tries.

Ownership

A read is readable only with the key that started it. The minting key's id is stamped on the job when it starts.

A search started in the consumer app has no key id at all, which makes it unreadable through the API entirely -- not merely unreadable by the wrong key. The absence of an owner is a closed door, not an open one.

Upstream failure

Three services do the work: the vision provider, the extraction service, the search provider. An outage or a rate limit at any of them surfaces as a failed read, and a failed read is refunded.

Inside the job, a stage retries on its own -- that is what the step boundaries in chapter 3 are for. What reaches you as failed has already been retried and is either a stable verdict or a genuine outage. One retry from your side is reasonable; two is a loop.

Timeouts are not failures

A read that is still running is still running. The request id stays valid for the job's retention window,

and collecting it later is free.

If a client gives up waiting, the correct move is to keep the id and collect it later, never to start again -- starting again is another charged request for the same photograph.

9. The honesty contract

Objects are spotted, not identified. Results are visually similar, not the same product.

This chapter exists because that is not a style guide. It is the only vocabulary the architecture can support, and the reason is in chapter 1: there is no verification step and no similarity score anywhere in the pipeline.

What is actually known

A cutout is what the reading model reported, rendered by an editing model. Nothing compared it back to the original photograph.

A candidate is what visual search returned for that cutout, in its own order. Nothing measured how close it is.

So the true statement is "the reader saw something it called a floor lamp, and visual search returned these products for the picture we made of it." Every shorter phrasing is either that, or a claim.

The words, and what they would commit to

Do not say	Because it claims
-----	-----
identified, detected, recognised	a verification step ran
found it, this is it, exact	the product was matched
match, best match	a similarity score exists
94% similar	a number was computed
guaranteed, verified	someone checked
Say	Which is true
-----	-----
spotted	the reader reported it
visually similar	search returned it for the cutout
a lead, worth checking	you should open the retailer page
nothing clearly enough visible to shop for	the honest empty result

Why this is a technical constraint

Two reasons, and neither is about tone.

The first is that the failure is invisible. A wrong cutout is a well-lit, centred, plausible product photograph of the wrong thing. It passes every check you would think to write -- one object, white background, no person. Only a human comparing it to the source notices. A product that says "identified" is therefore asserting something that nothing in the system, and nothing in your integration, is in a position to detect when it is false.

The second is that the claim transfers. If your UI says found, a customer who orders the wrong sofa is not wrong to be annoyed, and the promise they were given was yours. The pipeline gave you

evidence; the claim was added downstream.

What this looks like in practice

The consumer app presents every object as spotted and every result as visually similar, calls a try-on render a preview rather than a fitting, and keeps the retailer visible on every product tile so an outbound click is never a surprise. When a search finds nothing, it says the photo had nothing clearly shoppable rather than showing an empty grid.

The published agent skill makes the vocabulary an explicit rule, because a model summarising a result set will reach for found unprompted.

If you want to make a stronger claim

Build the thing that justifies it. A verification pass that compares each cutout to the source region, or a scoring pass over candidates, would earn stronger words -- and it would be your pass, your threshold and your liability.

Until then the honest sentence is short, and it is enough: here is what we spotted, here is what looks like it, go and look.